Applying the SCR Requirements Method to a Simple Autopilot[*]

In Proc. *Fourth NASA Langley Formal Methods Workshop, Sep 1997*

Ramesh Bharadwaj and Constance Heitmeyer
Center for High Assurance Computer Systems (Code 5546)
Naval Research Laboratory
Washington, DC 20375
{ramesh,heitmeyer}@itd.nrl.navy.mil
http://www.itd.nrl.navy.mil/ITD/5540/personnel/heitmeyer.html

Abstract

Although formal methods for developing computer systems have been available for more than a decade, few have had significant impact in practice. A major barrier to their use is that developers find formal methods difficult to understand and apply. One exception is a formal method called SCR for specifying computer system requirements which, due to its easy-to-use tabular notation and demonstrated scalability, has achieved some success in industry.

To demonstrate and evaluate the SCR method and tools, we recently used SCR to specify the requirements of a simplified mode control panel for the Boeing 737 autopilot. This paper presents the SCR requirements specification of the autopilot, outlines the process we used to create the SCR specification from a prose description, and discusses the problems and questions that arose in developing the specification. Formalizing and analyzing the requirements specification in SCR uncovered a number of problems with the original prose description, such as incorrect assumptions about the environment, incompleteness, and inconsistency. The paper also introduces a new tabular format we found useful in understanding and analyzing the required behavior of the autopilot. Finally, the paper compares the SCR approach to requirements with that of Butler [5], who uses the PVS language and prover [14] to represent and analyze the autopilot requirements.

1 Introduction

Although formal methods for developing computer systems have been available for more than a decade, few of these methods have had significant impact in the development of practical systems. A major impediment to the use of formal methods in industrial software development is the widespread view that the methods are impractical. Not only do developers regard most formal methods as difficult to understand and apply; in addition, they have serious doubts about the scalability and cost-effectiveness of the methods.

A promising approach to overcoming these problems is to hide the logic-based notation associated with most formal methods and to adopt a notation, such as a graphical or tabular notation, that developers find easy to use. Specifications in the more "user-friendly" notation can be translated automatically to a form more amenable to formal analysis. In addition, the formal method should be supported by powerful, easy-to-use tools. To the extent feasible, the tools should detect software errors automatically and provide easy-to-understand feedback useful in tracing the cause of an error.

By providing a "user-friendly" tabular notation with demonstrated scalability, a formal method called SCR for specifying the requirements of computer systems has already achieved some success in practice. Since the publication more than 15 years ago of the requirements specification for the A-7 aircraft's Operational Flight Program (OFP) [12, 1], many industrial organizations, including Rockwell-Collins, Lockheed, Grumman, and Ontario Hydro, have used SCR to specify requirements. To support the SCR method, we have recently developed a formal state machine model to define the SCR semantics [9, 11] and a set of integrated software tools to support validation and verification of SCR requirements specifications [8, 10, 4]. The tools include an *editor* for creating and modifying a requirements specification, a *simulator* for symbolically executing the specification, a *consistency checker* which checks the specification for well-formedness (e.g., syntax and type correctness, no missing cases or unwanted nondeterminism), and a *verifier* based on model checking for analyzing the specification for application properties.

[*] This work was supported by the Office of Naval Research.

To demonstrate and evaluate the SCR method and tools, we recently used SCR to specify the requirements of a simplified mode control panel for the Boeing 737 autopilot based on a description in a report by Butler [5]. Butler initially presents an incomplete prose description of the autopilot, and then adds prose to clarify the description. He also represents the required behavior in the PVS language [14] and verifies certain properties of the model using the PVS prover. This paper outlines the process we used to create the SCR requirements specification of the mode control panel, presents the SCR specification, and discusses the problems and questions that arose in developing the specification. Formulating the requirements specification in SCR exposed a number of problems with the prose description of the requirements, such as missing initial values, missing type definitions, missing units of measurement, lack of specificity, incorrect requirement, and several instances of inconsistency. The paper also introduces a new tabular format we found useful in understanding and analyzing the behavior of the autopilot. Finally, the paper compares the SCR approach to requirements with Butler's PVS approach.

2 The SCR Method: Background

2.1 SCR and Other Approaches

A recent article by Shaw [16] presents and discusses a number of different "specifications" of an automobile cruise control system. Each is constructed to satisfy different objectives. For example, Atlee and Gannon use a language based on logic to model the required behavior of a cruise control system [3] and a model checker to detect violations of selected properties. Below, we refer to their logic-based description as an *abstract model.*

The abstract model in [3] differs from an SCR specification in an important respect—namely, in the specific information it contains about the required behavior. Because its purpose is verification, the abstract model omits many details. For example, it does not describe the system outputs. Omitting this information in an abstract model is appropriate because the properties analyzed in [3] are independent of the system outputs and because a model useful in verification should only include information needed to reason about selected properties. Eliminating irrelevant information is especially important in verification. Without dramatic reductions in the size of the state space to be analyzed, model checking is usually infeasible. Moreover, the elimination of irrelevant facts is also beneficial in mechanical theorem proving where the model to be analyzed should only include those facts needed to establish the properties of interest.

In contrast to the abstract model of the system described in [3], the SCR requirements specification is a repository for all information that developers will need to construct the system software. Hence, it is necessarily more detailed and less abstract than a model useful in verification. An advantage of the SCR approach to requirements is that it not only provides detailed guidance on exactly what information belongs in a requirements document, but in addition provides a conceptual model of the system to be developed as well as special language constructs for representing the system requirements. This detailed guidance, system model, and language constructs specialized for requirements specification are lacking in more generic languages such as Statecharts [7] and PVS which, unlike SCR, are not customized for requirements specification.

2.2 The SCR Model

In the SCR approach, the system requirements are specified as a set of relations that the system must maintain between quantities of interest in its environment. In SCR, a requirements specification provides a "black box" description of the required behavior as two relations, REQ and NAT, from *monitored variables*, representing environmental quantities that the system monitors, to *controlled variables*, representing environmental quantities that the system controls [15]. NAT describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ describes the relation the system must maintain between the environmental quantities represented by the monitored and controlled variables. In SCR, these relations are specified concisely using a tabular notation.

To provide a precise and detailed semantics for the SCR method, the SCR model represents a system as a finite state automaton and describes the monitored and controlled variables and other constructs that make up an SCR specification in terms of that automaton [11, 9]. To concisely describe the required relation between the monitored and controlled variables, the model uses four constructs—modes, terms, conditions, and events. A *mode class* is a partitioning of the system states. Each equivalence class in the partition is called a *system mode* (or simply *mode*). A *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on a system state. An *event* occurs when the value of any system variable changes (a system variable is a monitored or controlled variable, a mode class, or a term). The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \overset{\mathrm{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed conditions c and d are evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. Informally, this denotes the event "predicate c becomes *true* in the new state when predicate d holds in the old state". The notation "`@F(c)`" denotes the event `@T(NOT c)`. During the operation of the system, the environment changes a monitored variable, causing an *input event*. In response, the system updates terms and mode classes and changes controlled variables.

3 Developing the SCR Requirements

Figure 1 illustrates the simplified mode control panel for the Boeing 737 as described in [5]. The autopilot monitors the aircraft's altitude (ALT), flight path angle (FPA) and calibrated air speed (CAS) and controls three displays which, depending on the mode, show either the current or desired value of the aircraft's altitude, its flight path angle, and its airspeed. The pilot enters (i.e., preselects) a new value into a display by using one of three knobs next to the displays and engages or disengages the autopilot by pressing one of four buttons at the top of the panel. Appendix A contains a prose description of the system adapted from [5]. The reader should note that the prose presented by Butler in [5] was intended as an example and is therefore (intentionally) incomplete. In the prose presented in this paper, we have (to the best of our knowledge) eliminated all intended incompleteness. Also, the variable names have been changed slightly to conform to the naming conventions of SCR specifications.

3.1 Environmental Variables

In the autopilot specification, we use the prefix "`m`" to indicate the names of monitored variables. The *type* of a monitored variable indicates the range of values that may be assigned to the variable. The autopilot system monitors the current altitude (represented by monitored variable `mALTcurrent`), the current flight path angle (`mFPAcurrent`), and the current calibrated air speed (`mCAScurrent`). Each of these monitored variables is of type integer. We assume that the autopilot measures the altitude `mALTcurrent` in `feet`, the flight path angle `mFPAcurrent` in `degrees`, and the calibrated air speed `mCAScurrent` in `feet per second`. The monitored variables `mALTsw`, `mATTsw`, `mCASsw`, and `mFPAsw` represent the positions of the four buttons, and each is either `on` or `off`. Finally, the monitored variables `mALTdesired`, `mCASdesired`, and `mFPAdesired` represent the values indicated by the three knobs and range over the integers.

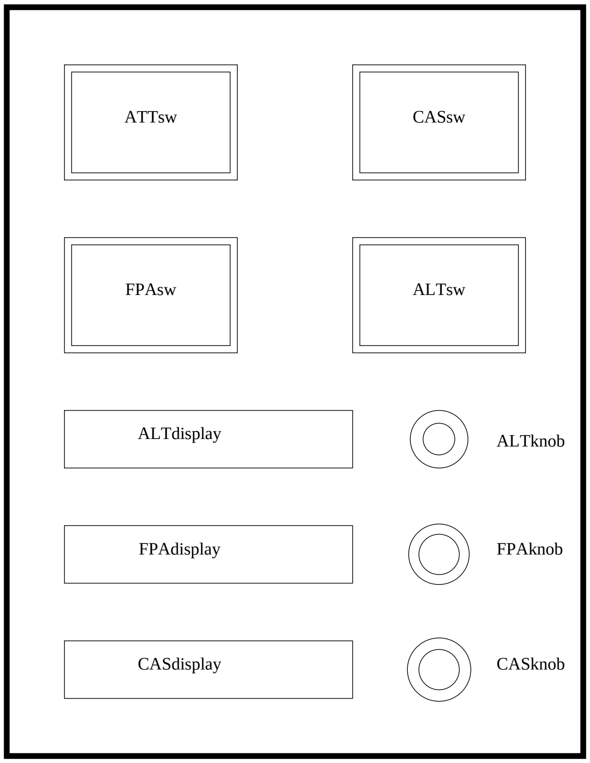


Figure 1: Mode Control Panel

The controlled variables are assigned names with the prefix "`c`". Just as for monitored variables, each controlled variable has an assigned type. As in [5], we only model the mode control panel, omitting commands sent to the flight control computer. We represent the three controlled quantities of the mode control panel as `cALTdisplay`, `cFPAdisplay`, and `cCASdisplay` and assume each is of type integer. We further assume that `cALTdisplay` displays the altitude in `feet`, `cFPAdisplay` displays the flight path angle in `degrees` and `cCASdisplay` displays the calibrated air speed in `feet per second`.

3.2 System Modes

The SCR specification includes a single mode class `mcStatus` containing modes in the set {`ALTmode`, `ATTmode`, `FPAarmed`, `FPAunarmed`}. When the system is in FPA mode and the altitude engage mode is "armed", we say the system is in mode FPAarmed. When the system is in the "normal" flight path angle selected mode, we say the system is in mode FPAunarmed. Thus, the system is in FPAmode when mcStatus is either FPAarmed or FPAunarmed. Because the system can be in the calibrated air speed mode independently of whether it is in ALTmode, ATTmode, or FPAmode, we exclude the calibrated air speed mode from mcStatus and use a term to describe whether the system is in this mode (see below).

3.3 Terms

Terms are assigned names with the prefix "t". The autopilot specification contains five terms, each of type boolean. The terms tALTpresel, tCASpresel, and tFPApresel indicate whether the pilot has preselected the altitude, the calibrated air speed, or the flight path angle using one of the three knobs. The term tNear denotes when the difference between the desired altitude and the current altitude is less than or equal to 1200 feet, i.e., $\texttt{mALTdesired} - \texttt{mALTcurrent} \leq 1200$. Finally, the term tCASmode indicates whether the system is in the calibrated air speed mode.

3.4 Relation REQ

The relation REQ is specified by a set of tables, one for each controlled variable, term, and mode class.

Mode Transition Table. The *mode transition table* in Figure 2 specifies the behavior of the mode class mcStatus. In the table, the expression CHANGED(x) denotes the event "variable x has changed value". The table defines all events that change the value of the mode class mcStatus. For example, the fourth row of the table states: "If mcStatus is ALTmode, and mATTsw is switched on, or the setting of knob mALTdesired is changed, then mcStatus changes to ATTmode." An assumption is that events omitted from the table do not change the value of the mode class. For example, when the system is in ALTmode, pressing the button labeled "ALTsw" (that is, the occurrence of the input event @T(mALTsw=on)) does not change the value of mcStatus.

Each row in the mode transition table in Figure 2 corresponds to certain parts of the prose description in Appendix A. We describe this correspondence below by associating each row of the table with the number of a paragraph in the prose description in Appendix A. In some cases, two rows of the table are derived from the same paragraph; for example, rows R3 and R5 are both derived from paragraph 1.

R1 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., pressing ALTsw engages ALTmode. *However, the altitude must be preselected before* ALTsw *is pressed* (paragraph 4). *If the pilot dials an altitude that is more than* 1200 *feet above* ALTcurrent *and then presses* ALTsw, *then* ALTmode *will not directly engage* (paragraph 3).

R2 *If the pilot dials into* ALTdesired *an altitude that is more than* 1200 *feet above* ALTcurrent *and then presses* ALTsw, *then* ALTmode *will not directly engage. Instead, the altitude engage mode will change to "armed" and* FPAmode *is engaged* (paragraph 3).

R3 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., by pressing FPAsw the pilot engages FPAunarmed.

R4 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., pressing ATTsw should engage ATTmode, **OR** *if the pilot dials in a new altitude while* ALTmode *is engaged, then* ALTmode *is disengaged and* ATTmode *is engaged* (paragraph 7).

R5 Same as row R3.

R6 Combination of scenarios for rows (4) and (3) above.

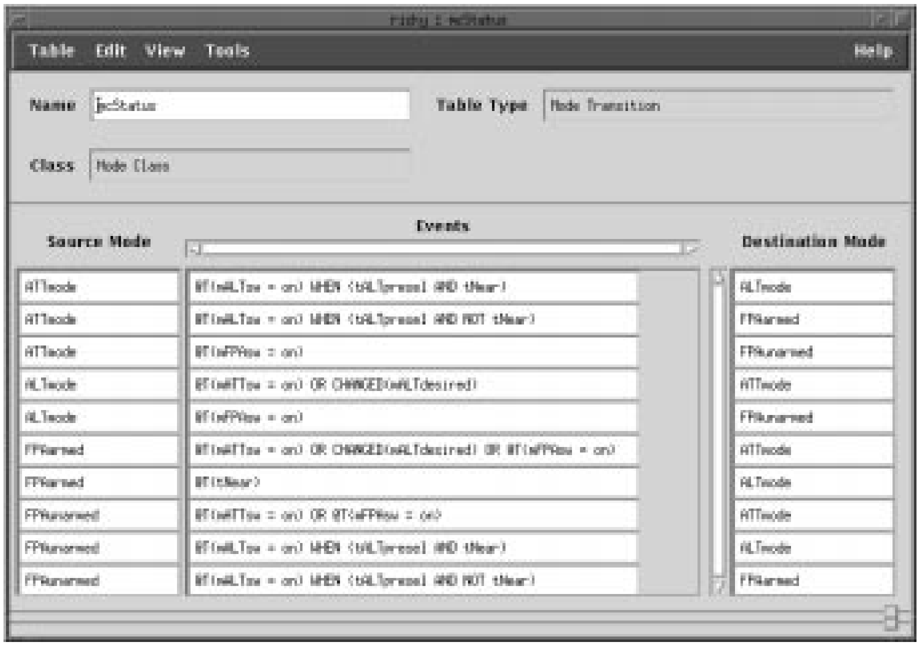
R7 FPAmode *will remain engaged until the aircraft is within* 1200 *feet of* ALTcurrent, *then* ALTmode *is automatically engaged* (paragraph 3).

R8 *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1), i.e., by pressing mATTsw the system enters ATTmode, **OR** FPAsw *toggles on and off every time it is pressed* (paragraph 5).

R9 Same as row R1.

R10 Same as row R2.

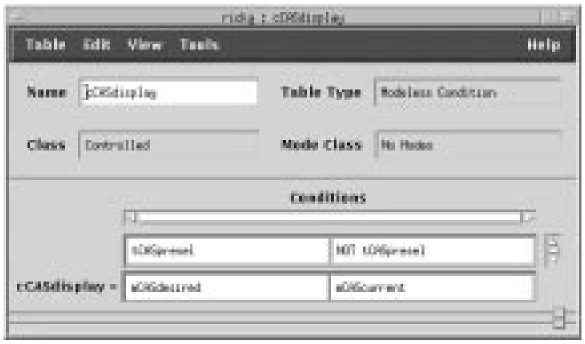
Term and Controlled Variable Tables. Each of the three terms, tALTpresel, tCASpresel, and tFPApresel, is *true* when the corresponding display, cALTdisplay, cCASdisplay, or cFPAdisplay, shows the "preselected" value and is *false* when the corresponding display shows the "current" value.



Name: mcStatus
Table Type: Mode Transition
Class: Mode Class

Source Mode	Events	Destination Mode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAarmed
ATTmode	@T(mFPAsw = on)	FPAunarmed
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAunarmed
FPAarmed	@T(mATTsw = on) OR CHANGED(mALTdesired) OR @T(mFPAsw = on)	ATTmode
FPAarmed	@T(tNear)	ALTmode
FPAunarmed	@T(mATTsw = on) OR @T(mFPAsw = on)	ATTmode
FPAunarmed	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
FPAunarmed	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAarmed

Figure 2: Mode Transition Table for mcStatus



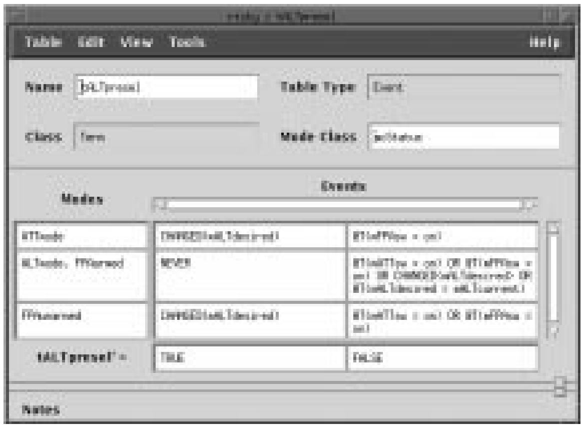
Name: cCASdisplay
Table Type: Modeless Condition
Class: Controlled
Mode Class: No Modes

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCAScurrent

Figure 3: Condition Table for cCASdisplay

Figure 3 is a *condition table* which specifies the behavior of the display cCASdisplay. This table states: "If tCASpresel is *true*, then cCASdisplay has the value mCASdesired; otherwise, it has the value mCAScurrent". The behavior of displays cALTdisplay and cFPAdisplay are specified similarly (see Appendix B).

The event table in Figure 4 specifies the behavior of tALTpresel. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. For example, the first entry in the first row states: "If mcStatus is ATTmode and mALTdesired changes value, then tALTpresel becomes *true*." The entry "NEVER" in an event table means that no event can cause the variable defined by the table to assume the value in the same column as the entry; thus, the entry "NEVER" in row 2 of the table means, "When mcStatus is ALTmode or FPAarmed, then no event can cause tALTpresel to become *true*." Figure 5 shows the event table for tFPApresel.



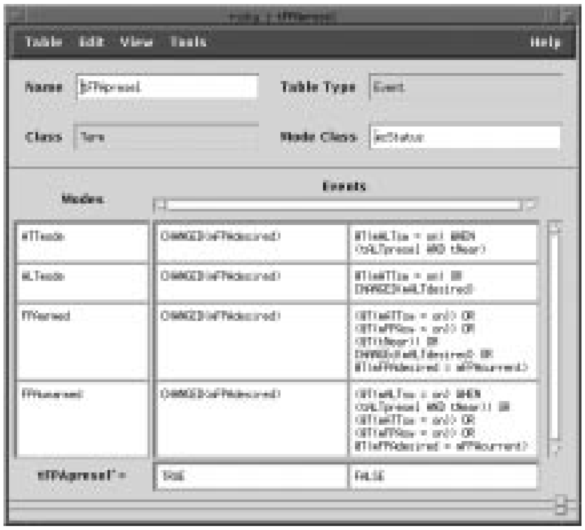
Name: tALTpresel
Table Type: Event
Class: Term
Mode Class: mcStatus

Modes	Events	
ATTmode	CHANGED(mALTdesired)	@T(mFPAsw = on)
ALTmode, FPAarmed	NEVER	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) OR @T(mALTdesired = mALTcurrent)
FPAunarmed	CHANGED(mALTdesired)	@T(mATTsw = on) OR @T(mFPAsw = on)
tALTpresel' =	TRUE	FALSE

Figure 4: Event Table for tALTpresel

4 Questions and Issues

Developing the SCR specification raised a number of questions about the required behavior of the mode control panel described by the prose in [5]. In a few cases (noted below), the problems were corrected in Butler's PVS formulation. These questions arose because applying the SCR method exposes instances of incompleteness and inconsistency



Name: tFPApresel
Table Type: Event
Class: Term
Mode Class: mcStatus

Modes	Events	
ATTmode	CHANGED(mFPAdesired)	@T(mALTsw = on) WHEN (tALTpresel AND tNear)
ALTmode	CHANGED(mFPAdesired)	@T(mATTsw = on) OR CHANGED(mALTdesired)
FPAarmed	CHANGED(mFPAdesired)	(@T(mATTsw = on)) OR (@T(mFPAsw = on)) OR (@T(tNear)) OR CHANGED(mALTdesired) OR @T(mFPAdesired = mFPAcurrent)
FPAunarmed	CHANGED(mFPAdesired)	(@T(mALTsw = on) WHEN (tALTpresel AND tNear)) OR (@T(mATTsw = on)) OR (@T(mFPAsw = on)) OR @T(mFPAdesired = mFPAcurrent)
tFPApresel' =	TRUE	FALSE

Figure 5: Event Table for `tFPApresel`

in the specification. This section presents these problems and describes how we resolved them. In resolving each problem, we made an educated guess about the actual requirement. Appendix B contains our revised SCR specification. For this specification to be acceptable, however, our decisions would need to be reviewed by system engineers with expert knowledge of the Boeing 737 autopilot.

4.1 Incompleteness

Developing the SCR specification exposed numerous instances of incompleteness in the prose description. First, the prose provides no information about the types, ranges, and units of measurement of the monitored variables that represent the current and desired altitude, (`mALTcurrent` and `mALTdesired`), the current and desired flight path angle (`mFPAcurrent` and `mFPAdesired`), and the current and desired calibrated air speed (`mCAScurrent` and `mCASdesired`). In addition, the ranges and units of measurement of the three controlled variables `cALTdisplay`, `cFPAdisplay`, and `cCASdisplay` are also omitted. This information is also missing in Butler's PVS model due to its abstract level. As noted above, the SCR specification represents these quantities as integers. In the final specification, even more precise information about the types would be required. For example, what are the minimum and maximum values of each monitored quantity? Are negative integers acceptable?

In addition, the prose description does not indicate the initial state of the autopilot. Butler's PVS description, however, does provide this information. Our SCR specification states that in the initial state `mcStatus` is `ATTmode`, `tCASmode = false`, the desired and current altitude are 0, etc. Appendix B shows the initial values and types of the monitored and controlled variables, terms, and mode class in the SCR specification. (In a specification produced by our toolset, the variable and mode class dictionaries, omitted here due to space limitations, would contain this information.)

4.2 Lack of Specificity

Paragraph 5 of the prose description states: "`FPAsw` *toggles on and off every time it is pressed.*" Paragraph 1 states: "*One of the three modes* `ATTmode`*,* `FPAmode`*, or* `ALTmode` *should be engaged at all times.*" From these two sentences one can infer that if `FPAmode` is toggled off by `FPAsw`, then one of `ATTmode` or `ALTmode` is engaged; *but not which one.* In our specification as well as Butler's, the decision is to engage `ATTmode`.

4.3 Wrong Interpretation

Paragraph 3 of the prose description states: "*If the pilot dials an altitude that is more than* 1200 *feet above* `ALTcurrent` *and then presses* `ALTsw`*, then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to* armed *and FPAmode is engaged.*" This sentence is either incorrect, or is misinterpreted by Butler—the PVS model in [5] sets the altitude engage mode to *armed* also in the case where the pilot dials a value for desired altitude that is more than 1200 feet *below* `ALTcurrent` and then presses `ALTsw`.

4.4 Incorrect Requirement

Paragraph 6 of the prose description states: "*Whenever a mode other than* `CASmode` *is engaged, all other preselected displays should return to current.*" However, consider the scenario where `CASmode` is engaged, `CASdisplay` shows the preselected value, and the pilot engages `ATTmode`. Clearly, returning `CASdisplay` to show the current value would be wrong in this situation since `CASmode` remains engaged. Therefore, the above sentence should read instead, "*Whenever a mode other than* `CASmode` *is engaged,* `ALTdisplay` *and* `FPAdisplay`*, if preselected, should return to current.*" The PVS model does the right thing for this scenario; however, Butler does not point out the error in the prose.

4.5 Inconsistent Requirement

Paragraph 6 of the prose description states: "*Whenever a mode other than* `CASmode` *is engaged,* ALTdisplay *[...], if preselected, should return to current.*" For the scenario of paragraph 7 "*If the pilot dials in a new altitude while* ALTmode *is engaged or the altitude engage mode is "armed", then* ALTmode *is disengaged and* ATTmode *is engaged.*" This suggests that tALTpresel is set to *false* (because ATTmode is engaged). On the other hand, the sentence "*However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display*" of paragraph 2 suggests that tALTpresel is set to *true* for this scenario (because the pilot has dialed-in a new altitude). We resolved this inconsistency by setting tALTpresel to *false*. Butler does not point out this inconsistency in the prose. Unlike the SCR specification, his PVS model resolves the inconsistency by preselecting the display.

5 Consistency Checking, Simulation

After creating the requirements specification, we used our automated consistency checker [8, 9] to check for proper syntax, type correctness, missing cases, nondeterminism, and other application-independent properties. Then, we used our simulator to symbolically execute the requirements specification [10] to ensure that the specification captures (what we assume is) the "customers' intent". For the autopilot specification, our consistency checker detected three instances of inconsistent requirements. Whereas we detected the inconsistency described in Section 4.5 by inspection, we overlooked the following three cases of inconsistency. Butler's PVS model, being too abstract, failed to detect any of these problems.

1. Consider the two sentences in paragraph 2 of the prose description: (1) *However, the pilot can enter a new value into* [FPAdisplay] *by dialing in the value using the knob* [FPAdesired] and (2) *Once the target value is achieved* [...], *the display reverts to showing the "current" value.* These sentences are inconsistent in the situation where the pilot enters a new value into FPAdisplay that is the same as FPAcurrent. In this situation, should the display show the dialed-in value or the current value?

   Butler[1] answers this question as follows: *The phrase ... "will this affect the preselected value (i.e., change it to current)" is difficult to interpret. I assume you meant "will this affect the status of the corresponding display (i.e., change it to current)". Interestingly in this case the status distinction is an artifact of the formalism because the target and current are the same value. So the "status" is merely a matter of choice/taste.*

   While we agree with Butler that the status distinction does not affect the *current* value of the display, we note that it does affect the *future* values displayed. For instance, suppose FPAcurrent proceeds to diverge from FPAdesired immediately following the above scenario. The sentence marked (1) specifies FPAdisplay to continue to show FPAdesired, whereas the sentence marked (2) specifies that FPAdisplay should track the "current" value. We resolved this issue by assuming that sentence (2) takes precedence over sentence (1). The table defining tFPApresel in Appendix B reflects this decision.

2. A similar scenario may be constructed for the calibrated air speed display. We resolved the issue in the same way as above. The table defining tCASpresel in Appendix B reflects this decision.

3. The first sentence of paragraph 7 states: *If the pilot dials in a new altitude while* [...] *the altitude engage mode is "armed", then* ALTmode *is disengaged and* ATTmode *is engaged.* However, dialing in a new altitude in mode FPAarmed can cause tNear to simultaneously become *true*, which leads to inconsistency. We ignore the event @T(tNear) in this situation. The revised specification for mcStatus in Appendix B reflects this decision.

6 Application Properties

After applying the consistency checker and the simulator, we wanted to check the requirements specification for critical application properties, such as safety properties. Verification may be carried out using an interactive theorem prover such as PVS [14, 5], or by using "lightweight" analysis tools such as model checkers. The SCR toolset supports proof of safety properties of requirements specifications using model checking based on state exploration [4]. The following sentences in paragraph 1 of the prose description are examples of properties of the autopilot mode control panel:

1. Only one of the three modes ALTmode, ATTmode, or FPAmode can be engaged at any time.

[1] Private communication.

2. One of the three modes, ALTmode, ATTmode, or FPAmode should be engaged at all times.

3. Engaging any of the three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.

4. The mode CASmode can be engaged at the same time as any of the other modes.

The type definition of mode class mcStatus is {ALTmode, ATTmode, FPAarmed, FPAunarmed}, and by definition, the system is in FPAmode if mcStatus is FPAarmed or FPAunarmed. We denote the system being in CASmode by the boolean term tCASmode whose value is independent of mcStatus. By this choice of the domain for mode class mcStatus, and the definition of tCASmode, the above properties are trivially satisfied (and verified automatically by a type checker).

5. Whenever the altitude engage mode is "armed", FPAmode is engaged.

For the SCR autopilot specification, this follows directly from the definition of FPAmode, i.e., the system is in FPAmode if mcStatus is FPAarmed or FPAunarmed. Therefore, if mcStatus is FPAarmed, the system is in FPAmode.

We used the Spin model checker to verify the requirements specification for two additional properties stated in [5] which are listed below. These properties could not be checked using simple type checking.

P1 When FPAmode is disengaged, the FPA display reverts to showing the "current" value.

P2 When ALTmode is disengaged, the ALT display reverts to showing the "current" value.

We currently check two classes of properties: *state invariants*, which assert the truth of a predicate formula for all reachable states of a system, and *transition invariants*, which assert the truth of a predicate formula (on two states) for all pairs of consecutive states of a system. Both P1 and P2 are transition invariants. Model checking can be ineffective in practice due to *state explosion*. By their very nature, the number of reachable states of practical systems is usually very large in relation to their logical representation. Several techniques have been proposed in the literature for limiting state explosion. The technique we use is *abstraction*—instead of model checking the whole SCR specification, we model check a smaller, more abstract model. To obtain the abstraction, we exploit the structure of the formula and the structure inherent in all SCR specifications. We use the two correctness preserving reductions of [4] to derive the abstract specification. Finally, we translate the abstract specification to PROMELA, the language of Spin [13], and run Spin on the PROMELA model.

When we initially attempted to check property P1, Spin detected a violation. The counterexample generated by Spin had 4867 states (which translates to 811 SCR "steps"). The shortest counterexample had only 73 states (12 SCR "steps"). By running the counterexample through the simulator of our toolset, we were able to pinpoint the cause of the property violation—a typographical error in the mode transition table for mcStatus.

7 A New Tabular Format

In [5], Butler presents two different PVS models of the panel requirements. In his initial model, he identifies the different input events (e.g., changing the ALT button to on, setting the knob labeled ALT to a new value, etc.) and then specifies the required behavior by describing the state changes and the changes in the displays that each input event causes. Thus, his initial model is organized by the input events. His second model defines a set of modes and describes the required behavior in terms of those modes. Butler claims that the latter organization, which is the organization used in SCR specifications, results in "a more complex formulation for this example problem." Moreover, he notes that his initial model is smaller, containing 373 words in contrast to 761 words.

We agree that understanding the overall system behavior can be difficult when that behavior is specified in numerous tables. This problem is overcome to some extent by the dependency graph produced by our toolset (see Appendix B), which shows all the variables in the specification and their dependencies. In initially creating the SCR specification, we developed individual tables for the mode class, the terms, and the three control variables. To enhance our understanding, we introduced a new tabular format that combines several smaller tables into a larger table. This larger table, which specifies the values of several variables each defined by either a mode transition table or an event table, is similar to the "selector table" used in the original A-7 requirements specification[2].

[2] A similar tabular format was also used in Lockheed's SCR specification of the OFP for the C-130J aircraft.

The specification in Appendix B contains two examples of the new tabular format. The first example defines the values of the terms tCASmode and tCASpresel. The other defines the values of the mode class mcStatus and the terms tALTpresel and tFPApresel. The new tabular format is useful because it collects in a single place all events that change a set of related variables. For example, the table defining mode class mcStatus and the terms tALTpresel and tFPApresel shows that when the system is in mode FPAarmed, pressing either mATTsw or mFPAsw causes the system to enter ATTmode and sets both tALTpresel and tFPApresel to *false*. Although the new table format is useful, the three separate tables defining mcStatus, tALTpresel, and tFPApresel are also useful, but for answering a different set of questions: for example, the table defining the mode class mcStatus identifies all events that can change the current mode but does so without the clutter of the extra information in the new table. Further, in a large application, merging tables together into a single table could produce very large tables that would be difficult to understand. As a result, we plan to support both our original tables, which define individual variables, and the new table, which defines two or more variables—and to automatically generate the larger table from selected tables that define individual variables.

In our view, the SCR specification in Appendix B is both easy to understand and concise. The complete specification is contained in two pages. Moreover, the SCR specification has an advantage over Butler's PVS model: it is easier to change. For example, our initial version of the specification defined a term called tArmed and only three modes, FPAmode, ALTmode, and ATTmode, in the mode class mcStatus. Our revised version removed the term tArmed and replaced mode FPAmode with two modes FPAarmed and FPAunarmed, thus producing a specification that was more concise and easier to understand. Making the change was quite straightforward—we simply eliminated the table for tArmed, revised the table for mcStatus, and modified the tables for tALTpresel and tFPApresel to describe the required behavior in modes FPAarmed and FPAunarmed. The tables defining the displays as well as all rows of tables that did not involve FPAmode were unchanged.

8 SCR versus PVS

Although PVS was not specifically designed to specify requirements, Butler advocates the use of the PVS language and prover for requirements specification [5]. In his report [5], Butler presents two PVS models of the mode control panel and verifies properties of the model organized by inputs using the PVS prover. As noted in the introduction, different formal models serve different purposes. While the PVS model of the panel allowed Butler to verify certain properties of interest, in our view, PVS is not a good notation for expressing requirements specifications to be used by software developers. This is because PVS, a language based on higher-order logic, produces specifications that are less readable by practitioners than specifications in alternative, more user-friendly languages. Moreover, because PVS is not part of a requirements method but is a general-purpose language designed to specify mathematical models, most PVS models omit (abstract away) information needed in a requirements specification—that is, PVS models are usually incomplete. Many of the questions that arose in our development of the SCR specification emerged because an SCR specification requires information that is lacking in Butler's PVS model.

Below, we compare the SCR approach to requirements specification and verification with the approach used by Butler. Although some of the problems we discuss are intrinsic to PVS, others are the result of decisions Butler made in developing the PVS model.

8.1 What are the requirements?

In the SCR approach, a requirements specification is complete when the specification contains all the information software developers will need to design and implement the software. To accomplish this, the specification must identify the quantities of interest in the system's environment, in particular, the monitored and controlled quantities, and specify the required relation between them. Butler's PVS model does not clearly identify the environmental quantities of interest. Nor does the PVS model clearly delineate monitored and controlled quantities. The result is that one cannot infer from the PVS model the required relationships among these quantities.

The PVS model makes use of *actions* or *events* as undefined (primitive) elements. In SCR, in contrast, the system inputs and outputs are modeled as variables, thus capturing more semantic information about the system behavior. This semantic information can be exploited in analyzing the specification for errors. For example, the PVS model assumes that certain input events are mutually disjoint, which results in the omission of an input event from the model (see Section 8.2). Since the SCR specification explicitly defines the environmental quantities of interest, this incompleteness in the specification was automatically detected during consistency checking.

The requirements specification presented in Appendix B is closer than the PVS model to a "real" requirements specification useful to developers. We note, however, that it is still incomplete in at least three respects. First, the I/O devices (or subsystems) that the autopilot uses to measure and compute the monitored and controlled quantities must be specified. Second, the required timing and accuracy of the system is yet unspecified. Third, the constraints imposed by NAT need to be specified. Once these aspects of the required software behavior are provided, developers would have all the information necessary to design, implement, and test the system.

8.2 What are the constraints?

To be complete, a requirements specification (including ours) should model the constraints that physical laws and the system's environment impose on the environmental quantities. For example, changes in altitude are limited by physical laws (e.g., the laws of gravity) and by the maximum rate at which the Boeing 737 can gain or lose altitude. Inclusion of such constraints in the specification can be used later in software development to do sanity checks on the specification—and the code—and to indicate when some fault has occurred (e.g., a sensor measuring altitude has failed).

Relations NAT and REQ are specified separately in the SCR method (see Section 2.2). The PVS model, on the other hand, does not distinguish relationships that arise from existing physical or other constraints (relation NAT of SCR) and relationships that are to be enforced by the system (relation REQ of SCR). This gap leads to two related problems: *overspecification* and *incompleteness*.

Overspecification. It is useful (and simpler) to ignore impossible situations (i.e., situations ruled out by NAT) in the definition of relation REQ. For example, the PVS model defines the following events:

alt_reached	the altitude reaches the preselected value
alt_gets_near	the altitude is now near, but not equal to the preselected value

On page 12 of [5], Butler addresses the scenario where the system is in mode FPAarmed (i.e., predicate tNear is *false*) and the event alt_reached occurs without alt_gets_near occurring first. This situation is clearly ruled out by relation NAT (if tNear is *false*, the current altitude cannot reach the preselected value without tNear becoming *true* first). Therefore, when specifying relation REQ, one need not deal with the above scenario.

Incompleteness. The PVS model considers the following events to be mutually disjoint:

input_alt	the action of dialing a value using mALTdesired
alt_reached	the altitude reaches the preselected value
alt_gets_near	the altitude is now near, but not equal to the preselected value

But, these events are *not* disjoint. For example, the action of dialing a value (denoted by event input_alt) can simultaneously cause either of the other two events to occur. The PVS model fails to consider these cases.

8.3 What is the level of abstraction?

The PVS description of the autopilot may be viewed as an abstract model of the mode control panel. For example, the monitored quantity ALTcurrent is denoted abstractly by two boolean variables alt_reached and alt_gets_near; boolean variable input_alt abstractly denotes the pilot dialing in the desired altitude using knob ALTdesired; etc. Once the model is constructed, one can use deductive reasoning to check that the model satisfies specified properties of interest, such as the application properties described above. Although such an approach is a good way to detect errors in the system requirements, our claim is that such abstract models must be transformed into a more concrete *requirements specification*, such as the one we present in Appendix B. Without a requirements specification, one cannot determine the monitored and controlled quantities of interest, and the required relationship between them. If the correspondence between the abstract model and the requirements specification is informal (and the required relation REQ is never specified explicitly), developers may misinterpret the requirements.

8.4 Role of tool support

The major strength of PVS is verification: using PVS, a user can analyze a specification for complex properties. In another task, we have used PVS to detect serious errors in the specification of a hybrid, real-time system [2]. However, because PVS was not designed for specifying and analyzing require-

Problem Description	Location	Phase			Reference
		Formulating SCR Spec	Consistency Checking	Model Checking	
Missing initial values	prose	many			4.1
Missing ranges, types, and units of measurement	prose, PVS	many			4.1
Lack of specificity	prose	1			4.2
Incorrect requirement	prose	1			4.4
Inconsistent requirements	prose, PVS	1	3		4.5, 5
Transcription error	SCR			1	6
Wrong interpretation	PVS	1			4.3
Overspecification	PVS	1			8.2
Incompleteness	PVS	1			8.2

Table 1: Detected Problems

ments, it lacks a requirements method. Without such a method, users have little guidance in developing a requirements specification. In contrast, the SCR method has been specifically designed to produce precise and complete requirements specifications. In our experience, tools that support a specific conceptual model and method are more effective than general-purpose tools. If a formal model lacks a strong underlying method, the benefits of automation are likely to be minimal ([6] provides more details). Since the SCR method focuses on a limited class of systems and standardizes the conceptual model, the notation, and the process, significant automated tool support is possible.

9 Analysis

Table 1 summarizes the problems we detected in applying the SCR method to the autopilot specification and identifies the specifications in which each problem occurs. Some of the problems listed (missing initial values, lack of specificity, and incorrect prose) were corrected by Butler. All the other problems, except the typographical error, were additional problems detected when we applied the SCR method to Butler's (presumably correct) prose specification. All but four of them were detected in formulating the SCR specification. Of the remaining four problems, the typographical error was detected by model checking and the remaining three cases of inconsistency were detected by our consistency checker.

It is clear from Table 1 that merely specifying a system using the SCR method *without any automated analysis* can expose many problems. It may be argued that some of these problems might eventually be detected by the PVS prover. To do this, however, users must formulate properties that will expose the problems. It has been our experience that formulating *correct* properties for a large requirements specification can be non-trivial. Moreover, in our opinion, the effort needed to analyze the specification with PVS would be significantly greater than the effort involved in formulating the specification in SCR and analyzing it with the SCR tools.

Our experience can be compared to that of Miller[3], who reports that the SCR method helped uncover 18 errors in a specification of an autopilot at Rockwell-Collins. Of the errors, one-third of them were detected during formulation of the SCR specification, one-third by the consistency checker, and one-third during simulation. (Since we used the simulator to a much smaller degree than Miller, we found no significant errors using the simulator.) Our two efforts clearly demonstrate that lightweight methods are highly effective in uncovering errors in requirements specifications. It is also important to note that most of these errors, including those that were detected by formulating the specification in SCR, would have probably gone undetected without appropriate tool support.

10 Conclusions

In this paper, we outlined a process for creating an SCR requirements specification of a simplified mode control panel for the Boeing 737 autopilot, based on the prose description of the system presented in [5]. Developing an SCR specification of the autopilot uncovered a number of problems that were undetected in Butler's formalization of the problem using PVS. While PVS is useful for verifying deep properties of specifications, this study

[3] Private communication.

provides evidence that PVS is not well suited for formulating and analyzing requirements specifications, especially during the initial stages. This is due to a number of factors: the logic-based PVS language which software developers find difficult to apply, the mathematical sophistication and theorem proving skills that developers need to verify properties using the PVS prover, and the lack of a requirements method for PVS.

We envision a process for developing high-quality requirements specifications that combines the SCR technology and a mechanical prover, such as PVS. This process would rely on the light-weight SCR tools during the initial part of the requirements process—specification using a formal yet "user-friendly" notation to capture the requirements, automated consistency checking and model checking to detect violations of simple properties, and simulation to ensure that the specification captures the customers' intent. Once sufficient confidence in the specification is developed, a mechanical proof system, such as PVS, may be used to verify deep properties of the complete requirements specification or, more likely, safety-critical components. While software developers themselves will have the skills needed to apply the light-weight SCR tools, applying heavy-duty theorem proving is likely to require formal methods experts with the requisite mathematical sophistication and theorem proving skills.

Acknowledgments

We gratefully acknowledge Ricky Butler for providing helpful insights and for his prompt answers to all our questions about the autopilot mode control panel. We thank Todd Grimm and Bruce Labaw for implementing support for our verification method in the SCR toolset. We also acknowledge the very helpful comments of our colleagues Myla Archer and Bruce Labaw on earlier drafts of this paper.

References

[1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software Requirements for the A-7E Aircraft. Technical Report NRL-9194, NRL, Washington DC, 1992.

[2] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. *Proc. 1997 International Workshop on Hybrid and Real-Time Systems (HART'97)*, Grenoble, France, March 1997.

[3] J. M. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, pp 22–40, January 1993.

[4] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proc. First ACM SIGPLAN Workshop on Automatic Analysis of Software,* Paris, France, January 14, 1997.

[5] R. W. Butler. An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[6] S. R. Faulk. Software Requirements: A Tutorial. Technical Report *NRL/MR/5546-95-7775*, Naval Research Laboratory, Washington DC, 1995.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* 8(3), 231-274, June 1987.

[8] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc. 1995 Int'l Symposium on Requirements Engg.*, York, England, March 1995.

[9] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. on Software Engg. and Methodology*, 5(3), 231–261, July 1996.

[10] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc.* 12^{th} *Annual Conference on Computer Assurance*, NIST, Gaithersburg MD, June 1997.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report, Naval Research Laboratory, Washington DC, 1997. In preparation.

[12] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering* SE-6(1), Jan 1980.

[13] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[14] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In 11^{th} *International Conference on Automated Deduction*, LNCS-607, pp 748–752, 1992.

[15] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming,* 25(1), pp 41–62, Oct 1995.

[16] M. Shaw. Comparing architectural design styles. *IEEE Software*, November 1995.

A Description of the autopilot

1. *The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values, as shown in Figure 1. The system supports the following four modes: attitude control wheel steering (*`ATTmode`*), flight path angle selected (*`FPAmode`*), altitude engage (*`ALTmode`*), and calibrated air speed (*`CASmode`*).*

   Only one of the first three modes can be engaged at any time. The mode `CASmode` *can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes,* `ATTmode`*,* `FPAmode`*, or* `ALTmode` *should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.*

2. *There are three displays on the panel: altitude (*`ALTdisplay`*), flight path angle (*`FPAdisplay`*), and calibrated air speed (*`CASdisplay`*). The displays usually show the current values of altitude (*`ALTcurrent`*), flight path angle (*`FPAcurrent`*), and air speed (*`CAScurrent`*) of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display (*`ALTdesired`*,* `FPAdesired`*, or* `CASdesired`*). This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to* $25,000$ *feet, he will dial* $25,000$ *(using the knob* `ALTdesired`*) into* `ALTdisplay` *and then press* `ALTsw` *to engage* `ALTmode`*. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.*

3. *If the pilot dials into* `ALTdesired` *an altitude that is more than* $1,200$ *feet above the current altitude (*`ALTcurrent`*) and then presses* `ALTsw`*, then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged. The pilot must then dial in, using the knob* `FPAdesired`*, the desired flight-path angle into* `FPAdisplay`*, which will be followed by the flight-control system until the aircraft attains the desired altitude.* `FPAmode` *will remain engaged until the aircraft is within* $1,200$ *feet of* `ALTcurrent`*, then* `ALTmode` *is automatically engaged.*

4. `CASdesired` *and* `FPAdesired` *need not be pre-selected before the corresponding modes are engaged—the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before* `ALTsw` *is pressed. Otherwise, the command is ignored.*

5. `CASsw` *and* `FPAsw` *toggle on and off every time they are pressed. For example, if* `CASsw` *is pressed while the system is already in* `CASmode`*, that mode will be disengaged. However, if* `ATTsw` *is pressed while* `ATTmode` *is already engaged, the command is ignored. Likewise, pressing* `ALTsw` *while the system is already in* `ALTmode` *has no effect.*

6. *Whenever a mode other than* `CASmode` *is engaged, all other pre-selected displays should return to current.*

7. *If the pilot dials in a new altitude while* `ALTmode` *is engaged or the altitude engage mode is "armed", then* `ALTmode` *is disengaged and* `ATTmode` *is engaged. If the altitude engage mode is "armed" then* `FPAmode` *should be disengaged as well.*

B SCR Specification of the autopilot

Monitored Variables:

$\texttt{mALTcurrent}, \texttt{mCAScurrent}, \texttt{mFPAcurrent}$: Integer **initially** all 0;
$\texttt{mALTsw}, \texttt{mATTsw}, \texttt{mCASsw}, \texttt{mFPAsw}$: $\{\texttt{on}, \texttt{off}\}$ **initially** all $\texttt{off}$;
$\texttt{mALTdesired}, \texttt{mCASdesired}, \texttt{mFPAdesired}$: Integer **initially** all 0;

Controlled Variables:

$\texttt{cALTdisplay}, \texttt{cCASdisplay}, \texttt{cFPAdisplay}$: Integer **initially** all 0;

Terms:

$\texttt{tALTpresel}, \texttt{tCASpresel}, \texttt{tFPApresel}$: Boolean **initially** all $\texttt{false}$;
$\texttt{tCASmode}$: Boolean **initially** $\texttt{false}$;
$\texttt{tNear} \overset{\text{def}}{=} \texttt{mALTdesired} - \texttt{mALTcurrent} \leq 1200$;

Mode Class:

$\texttt{mcStatus}$: $\{\texttt{ALTmode}, \texttt{ATTmode}, \texttt{FPAarmed}, \texttt{FPAunarmed}\}$ **initially** $\texttt{ATTmode}$;

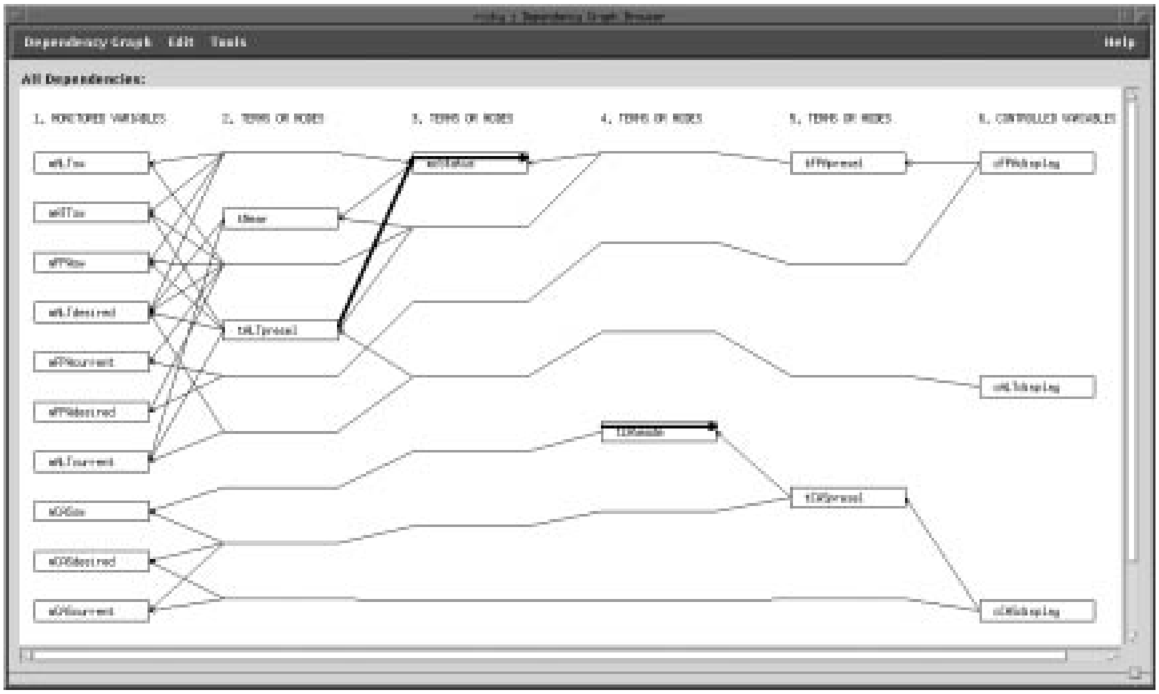


Figure 6: Variable Dependency Graph

Term = tCASmode			
	Events	tCASmode	tCASpresel
NOT tCASmode	@T(mCASsw=on)	true	
	CHANGED(mCASdesired)		true
tCASmode	@T(mCASsw=on)	false	false
	@T(mCASdesired=mCAScurrent)		false
	CHANGED(mCASdesired) AND mCASdesired' ≠ mCAScurrent'		true

Mode Class = mcStatus				
Old Mode	Events	New Mode	tALTpresel	tFPApresel
ATTmode	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired)			true
ALTmode	@T(mATTsw=on)	ATTmode	false	false
	@T(mFPAsw=on)	FPAunarmed	false	
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired)			true
	@T(mALTdesired = mALTcurrent)		false	
FPAarmed	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)	ATTmode	false	false
	CHANGED(mFPAdesired) AND mFPAdesired' ≠ mFPAcurrent'			true
	@T(tNear) AND mALTdesired = mALTdesired'	ALTmode		false
	@T(mFPAdesired = mFPAcurrent)			false
FPAunarmed	@T(mALTsw=on) WHEN (tALTpresel AND tNear)	ALTmode		false
	@T(mALTsw=on) WHEN (tALTpresel AND NOT tNear)	FPAarmed		
	@T(mATTsw=on) OR @T(mFPAsw=on)	ATTmode	false	false
	CHANGED(mALTdesired)		true	
	CHANGED(mFPAdesired) AND mFPAdesired' ≠ mFPAcurrent'			true
	@T(mFPAdesired = mFPAcurrent)			false

	Conditions	
	tALTpresel	NOT tALTpresel
cALTdisplay =	mALTdesired	mALTcurrent

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCAScurrent

	Conditions	
	tFPApresel	NOT tFPApresel
cFPAdisplay =	mFPAdesired	mFPAcurrent